ATTORNEY DOCKET NO.: 06502-0259

UNITED STATES PATENT APPLICATION

OF

ANN M. WOLLRATH,
ROBERT W. SCHEIFLER,
JAMES H. WALDO,
KENNETH C.R.C. ARNOLD,
ZANE PAN,
AND
BRIAN MURPHY

FOR

DYNAMIC LOOKUP SERVICE IN A DISTRIBUTED SYSTEM

DYNAMIC LOOKUP SERVICE IN A DISTRIBUTED SYSTEM

Related Applications

This application is a continuation-in-part of U.S. Patent Application No. 09/044,931, filed on March 20, 1998, which is incorporated by reference.

The following identified U.S. patent applications are relied upon and are incorporated by reference in this application.

U.S. Provisional Application No. 60/138,680, entitled "Jini™ Technology Helper Utilities and Services," bearing attorney docket no. 06502.6010-00000.

Provisional U.S. Patent Application No. 60/076,048, entitled "Distributed Computing System," filed on February 26, 1998.

U.S. Patent Application No. 09/044,923, entitled "Method and System for Leasing Storage," bearing attorney docket no. 06502.0011-01000.

U.S. Patent Application No. 09/044,838, entitled "Method, Apparatus and Product for Leasing of Delegation Certificates in a Distributed System," bearing attorney docket no. 06502.0011-02000.

U.S. Patent Application No. 09/044,834, entitled "Method, Apparatus and Product for Leasing of Group Membership in a Distributed System," bearing attorney docket no. 06502.0011-03000.

U.S. Patent Application No. 09/044,933, entitled "Method for Transporting Behavior in Event Based System," bearing attorney docket no. 06502.0054-00000.

U.S. Patent Application No. 09/044,919, entitled "Deferred Reconstruction of Objects and Remote Loading for Event Notification in a Distributed System," bearing attorney docket no. 06502.0062-01000.

U.S. Patent Application No. 09/045,652, entitled "Method and System for Deterministic Hashes to Identify Remote Methods," bearing attorney docket no. 06502.0103-00000.

U.S. Patent Application No. 09/044,790, entitled "Method and Apparatus for Determining Status of Remote Objects in a Distributed System," bearing attorney docket no. 06502.0104-00000.

U.S. Patent Application No. 09/044,930, entitled "Downloadable Smart Proxies for Performing Processing Associated with a Remote Procedure Call in a Distributed System," bearing attorney docket no. 06502.0105-00000.

U.S. Patent Application No. 09/044,917, entitled "Suspension and Continuation of Remote Methods," bearing attorney docket no. 06502.0106-00000.

U.S. Patent Application No. 09/044,835, entitled "Method and System for Multi-Entry and Multi-Template Matching in a Database," bearing attorney docket no. 06502.0107-00000.

U.S. Patent Application No. 09/044,839, entitled "Method and System for In-Place Modifications in a Database," bearing attorney docket no. 06502.0108.

U.S. Patent Application No. 09/044,945, entitled "Method and System for Typesafe Attribute Matching in a Database," bearing attorney docket no. 06502.0109-00000.

U.S. Patent Application No. 09/044,939, entitled "Apparatus and Method for Providing Downloadable Code for Use in Communicating with a Device in a Distributed System," bearing attorney docket no. 06502.0112-00000.

U.S. Patent Application No. 09/044,826, entitled "Method and System for Facilitating Access to a Lookup Service," bearing attorney docket no. 06502.0113-00000.

U.S. Patent Application No. 09/044,932, entitled "Apparatus and Method for Dynamically Verifying Information in a Distributed System," bearing attorney docket no. 06502.0114-00000.

U.S. Patent Application No. 09/030,840, entitled "Method and Apparatus for Dynamic Distributed Computing Over a Network," and filed on February 26, 1998.

U.S. Patent Application No. 09/044,936, entitled "An Interactive Design Tool for Persistent Shared Memory Spaces," bearing attorney docket no. 06502.0116-00000.

U.S. Patent Application No. 09/044,934, entitled "Polymorphic Token-Based Control," bearing attorney docket no. 06502.0117-00000.

U.S. Patent Application No. 09/044.915, entitled "Stack-Based Access Control," bearing attorney docket no. 06502.0118-00000.

U.S. Patent Application No. 09/044,944, entitled "Stack-Based Security Requirements," bearing attorney docket no. 06502.0119-00000.

U.S. Patent Application No. 09/044,837, entitled "Per-Method Designation of Security Requirements," bearing attorney docket no. 06502.0120-00000.

Field of the Invention

The present invention relates generally to data processing systems and, more particularly, to a dynamic lookup service in a distributed system.

Background of the Invention

A lookup service contains an indication of where network services are located within a distributed system comprised of multiple machines, such as computers and related peripheral devices, connected in a network (for example, a local area network, wide area network, or the Internet). A "network service" refers to a resource, data, or functionality that is accessible on the network. Typically, for each service, the lookup service contains an address used by a client (e.g., a program) to access the service (e.g., a printer).

Conventional lookup services are static: whenever updates to the lookup service are needed to either add a new service or delete an existing service, the lookup service is taken offline, rendering the lookup service inaccessible, and then, the lookup service is manually updated by the system administrator. During the time when the lookup service is offline, clients in the distributed system are unable to access the lookup service and any of its network services. Another limitation of conventional lookup services is that, when updated, clients are not made aware of the updates to the lookup service until they explicitly perform a refresh operation, which downloads the latest service information to the clients. Before such a refresh, however, if a client requests a service that is no longer available, an error occurs which may cause the client to hang. Also, before a refresh, the client is not aware of any new services that have been recently added to the lookup service. It is therefore desirable to improve lookup services for distributed systems.

Summary of the Invention

Systems consistent with the present invention provide an improved lookup service that allows for the dynamic addition and deletion of services. This lookup service allows for the update, i.e., addition and deletion of available services automatically, without user intervention. As a result, clients of the lookup service may continue using the lookup service and its associated services while the updates occur. Additionally, the lookup service provides a notification mechanism that can be used by clients to receive a notification when the lookup service is updated. By receiving such a notification, clients can avoid attempting to access a service that is no longer available and can make use of new services as soon as they are added to the lookup service.

In accordance with methods consistent with the present invention, a method is provided in a data processing system having a lookup service with associated services. This method receives a request by the lookup service for notification when the lookup service is updated, determines

when the lookup service is updated, and generates a notification when it is determined that the lookup service is updated.

In accordance with methods consistent with the present invention, a method is provided in a data processing system having a lookup service with associated services. This method sends a request to the lookup service to be notified when the lookup service is updated and receives an indication that the lookup service has been updated.

In accordance with methods consistent with the present invention, a method is provided in a data processing system having a lookup service with associated services and a client lookup manager with an associated cache. This method transmits an event by the lookup service that identifies a change to one of the associated network services. The client lookup manager receives the event and updates the associated cache to reflect the change.

In accordance with methods consistent with the present invention, a method is provided in a data processing system having a client and lookup service with associated services. This method receives a request from a client for access to a network service, then retrieves a reference from a cache reflecting a particular network service corresponding to the requested network service and transmits the reference to the client.

In accordance with systems consistent with the present invention, a data processing system comprising a memory and a processor is provided. The memory includes a lookup service containing indications of services that are available for use, a first client for updating the lookup service, and a second client for utilizing the lookup service while the first client is updating the lookup service. The processor runs the lookup service, the first client, and the second client.

In accordance with systems consistent with the present invention, a data processing system containing a memory and a processor is provided. The memory contains a lookup service with indications of services available for use and a client. The lookup service receives requests for notification of when the lookup service is updated, determines when the lookup service is updated, and generates notifications when the lookup service is updated. The client sends a request to the lookup service to be notified when the lookup service is updated. The processor runs the client and the lookup service.

In accordance with systems consistent with the present invention, a data processing system containing a memory and a processor is provided. The memory contains a lookup service with references to a plurality of network services available for use, a client, and a client lookup

manager with an associated cache stored on the client computer. The client lookup manager accesses the lookup service and stores service references in the cache. The processor runs the client and the lookup service.

In accordance with systems consistent with the present invention, a computer-readable memory device containing a data structure is provided. This data structure is for accessing a lookup service with associated network services available for use. The data structure contains a notify method for use by a client to register with the lookup service to receive a notification from the lookup service when the lookup service is updated.

Brief Description of the Drawings

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate an implementation of the invention and, together with the description, serve to explain the advantages and principles of the invention. In the drawings,

Figure 1 depicts a distributed system suitable for practicing methods and systems consistent with the present invention;

Figure 2 depicts a more detailed block diagram of a computer depicted in Fig. 1;

Figures 3A and 3B depict a flow chart of the steps performed when a client utilizes the lookup service depicted in Fig. 1;

Figure 4 depicts a flow chart of the steps performed by the lookup service of Fig. 1 when performing event-related processing;

Figure 5 depicts a conference room containing a number of devices consistent with the present invention;

Figure 6 depicts a screen displaying icons representing services available in the conference room of Figure 5;

Figure 7 depicts a screen displaying the available services provided by a computer in the conference room of Figure 5;

Figure 8 depicts an alternate embodiment of the computer depicted in Figure 2;

Figure 9 depicts a flow chart of the steps performed by the client lookup manager of Fig. 8 when populating the cache;

Figure 10 depicts a flow chart of the steps performed by the filtering algorithm contained within the client lookup manager;

Figure 11 depicts a flow chart of the steps performed by a first query processing algorithm contained within the client lookup manager;

Figure 12 depicts a flow chart of the steps performed by a second query processing algorithm contained within the client lookup manager; and

Figure 13 depicts a flow chart of the steps performed by a notification process contained within the client lookup manager.

Detailed Description of the Invention

Methods and systems consistent with the present invention provide an improved lookup service that allows for the dynamic addition and deletion of services. As such, the addition and deletion of services is performed automatically, without user intervention, and clients of the lookup service may continue using the services while the updates to the lookup service occur. Additionally, clients may register with the lookup service to receive notification of when the lookup service is updated. As a result, when an update occurs, all registered clients receive a notification of the update, enabling the clients to avoid attempting to access a service that is no longer available and to use a service recently added to the lookup service.

Overview of The Distributed System

Methods and systems consistent with the present invention operate in a distributed system ("the exemplary distributed system") with various components, including both hardware and software. The exemplary distributed system (1) allows users of the system to share services and resources over a network of many devices; (2) provides programmers with tools and programming patterns that allow development of robust, secured distributed systems; and (3) simplifies the task of administering the distributed system. To accomplish these goals, the exemplary distributed system utilizes the Java™ programming environment to allow both code and data to be moved from device to device in a seamless manner. Accordingly, the exemplary distributed system is layered on top of the Java programming environment and exploits the characteristics of this environment, including the security offered by it and the strong typing provided by it. The Java programming environment is more clearly described in Jaworski, Java 1.1 Developer's Guide, Sams.net (1997), which is incorporated herein by reference.

In the exemplary distributed system, different computers and devices are federated into what appears to the user to be a single system. By appearing as a single system, the exemplary

distributed system provides the simplicity of access and the power of sharing that can be provided by a single system without giving up the flexibility and personalized response of a personal computer or workstation. The exemplary distributed system may contain thousands of devices operated by users who are geographically disperse, but who agree on basic notions of trust, administration, and policy. Within the exemplary distributed system are various logical groupings of services provided by one or more devices, and each such logical grouping is known as a Djinn. A "service" refers to a resource, data, or functionality that can be accessed by a user, program, device, or another service and that can be computational, storage related, communication related, or related to providing access to another user. Examples of services provided as part of a Djinn include devices, such as printers, displays, and disks; software, such as applications or utilities; information, such as databases and files; and users of the system.

Both users and devices may join a Djinn. When joining a Djinn, the user or device adds zero or more services to the Djinn and may access, subject to security constraints, any one of the services it contains. Thus, devices and users federate into a Djinn to share access to its services. The services of the Djinn appear programmatically as objects of the Java programming environment, which may include other objects, software components written in different programming languages, or hardware devices. A service has an interface defining the operations that can be requested of that service, and the type of the service determines the interfaces that make up that service.

Fig. 1 depicts the exemplary distributed system 100 containing a computer 102, a computer 104, and a device 106 interconnected by a network 108. The device 106 may be any of a number of devices, such as a printer, fax machine, storage device, input device, computer, or other devices. The network 108 may be a local area network, wide area network, or the Internet. Although only two computers and one device are depicted as comprising the exemplary distributed system 100, one skilled in the art will appreciate that the exemplary distributed system 100 may include additional computers or devices.

Fig. 2 depicts the computer 102 in greater detail to show a number of the software components of the exemplary distributed system 100. One skilled in the art will appreciate that computer 104 or device 106 may be similarly configured. Computer 102 includes a memory 202, a secondary storage device 204, a central processing unit (CPU) 206, an input device 208, and a video display 210. The memory 202 includes a lookup service 212, a discovery server 214, and a

Java runtime system 216. The Java runtime system 216 includes the Java remote method invocation system (RMI) 218 and a Java virtual machine 220. The secondary storage device 204 includes a Java space 222.

As mentioned above, the exemplary distributed system 100 is based on the Java programming environment and thus makes use of the Java runtime system 216. The Java runtime system 216 includes the Java Application Programming Interface (API), allowing programs running on top of the Java runtime system to access, in a platform-independent manner, various system functions, including windowing capabilities and networking capabilities of the host operating system. Since the Java API provides a single common API across all operating systems to which the Java runtime system 216 is ported, the programs running on top of a Java runtime system run in a platform-independent manner, regardless of the operating system or hardware configuration of the host platform. The Java runtime system 216 is provided as part of the Java software development kit available from Sun Microsystems of Mountain View, CA.

The Java virtual machine 220 also facilitates platform independence. The Java virtual machine 220 acts like an abstract computing machine, receiving instructions from programs in the form of bytecodes and interpreting these bytecodes by dynamically converting them into a form for execution, such as object code, and executing them. RMI 218 facilitates remote method invocation by allowing objects executing on one computer or device to invoke methods of an object on another computer or device. Both RMI and the Java virtual machine are also provided as part of the Java software development kit.

The lookup service 212 defines the services that are available for a particular Djinn. That is, there may be more than one Djinn and, consequently, more than one lookup service within the exemplary distributed system 100. The lookup service 212 contains one object for each service within the Djinn, and each object contains various methods that facilitate access to the corresponding service. The lookup service 212 and its access are described in greater detail in co-pending U.S. Patent Application No. 09/044,826, entitled "Method and System for Facilitating Access to a Lookup Service," which has previously been incorporated by reference.

The discovery server 214 detects when a new device is added to the exemplary distributed system 100, during a process known as boot and join or discovery, and when such a new device is detected, the discovery server passes a reference to the lookup service 212 to the new device, so that the new device may register its services with the lookup service and become a member of the

Djinn. After the device discovers the lookup service 212, the device may access any of the services registered with the lookup service 212. Furthermore, the device may also advertise its own services by registering with the lookup service 212. Once registered, the services provided by the device may be accessed through the lookup service 212, by all other entities that also discover lookup service 212. The process of boot and join is described in greater detail in co-pending U.S. Patent Application No. 09/044,939, entitled "Apparatus and Method for providing Downloadable Code for Use in Communicating with a Device in a Distributed System," which has previously been incorporated by reference.

The Java space 222 is an object repository used by programs within the exemplary distributed system 100 to store objects. Programs use the Java space 222 to store objects persistently as well as to make them accessible to other devices within the exemplary distributed system. Java spaces are described in greater detail in co-pending U.S. Patent Application No. 08/971,529, entitled "Database System Employing Polymorphic Entry and Entry Matching," assigned to a common assignee, filed on November 17, 1997, which is incorporated herein by reference. One skilled in the art will appreciate that the exemplary distributed system 100 may contain many lookup services, discovery servers, and Java spaces.

Although systems and methods consistent with the present invention are described as operating in the exemplary distributed system and the Java programming environment, one skilled in the art will appreciate that the present invention can be practiced in other systems and other programming environments. Additionally, although aspects of the present invention are described as being stored in memory, one skilled in the art will appreciate that these aspects can also be stored on or read from other types of computer-readable media, such as secondary storage devices, like hard disks, floppy disks, or CD-ROM; a carrier wave from the Internet; or other forms of RAM or ROM. Sun, Sun Microsystems, the Sun Logo, Java, and Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries

The Lookup Service Definition

A lookup service provides a central registry of services available within a Djinn. This lookup service is the primary mechanism for programs to find services within the Djinn and is the foundation for providing user interfaces through which users and administrators can discover and interact with services of the Djinn.

The lookup service maintains a flat collection of service items. The collection is flat in that the service items do not form a hierarchy. Each service item represents an instance of a service available within the Djinn. The service item contains a stub (if the service is implemented as a remote object) or a serialized object (if the service is a local object stored in the lookup service) that programs use to access the service, and an extensible collection of attributes that describe the service or provide secondary interfaces to the service. A "stub" is code and data that facilitates access to a remote function, and a "serialized object" is an object placed in a serialized form.

Although the collection of service items is flat, a wide variety of hierarchical views can be imposed on the collection by aggregating items according to service types and attributes. The lookup service provides a set of methods to enable users and administrators to browse the collection and build a variety of user interfaces. Once an appropriate service is found, the user can interact with the service by loading a user interface applet, attached as another attribute on the item.

When a new service is created (e.g., when a new device is added to a Djinn), the service registers itself with the lookup service, providing an initial collection of attributes. For example, a printer may include attributes indicating speed (in pages per minute), resolution (in dots per inch), and whether duplex printing is supported. The attributes also contain an indicator that the service is new and needs to be configured. To configure a new service, the administrator locates an attribute that provides an applet for this purpose, and during configuration, the administrator may add new attributes, such as the physical location of the service and a common name for it.
The lookup service provides an event mechanism that generates notifications as new services are registered, existing services are deleted, or attributes of a service are modified. To use the event mechanism, a client registers to be notified upon the occurrence of a particular event, and when the event occurs, the lookup service notifies the client.

Programs (including other services) that need a particular type of service can use the lookup service to find an instance of the service. A match may be made based on the specific Java programming language types implemented by the service as well as the specific attributes attached to the service.

If a service encounters a problem that needs administrative attention, like a printer running out of toner, the service can add an attribute that indicates the problem. Administrators (or programs) can then use the event mechanism to receive notification of such problems.

The attributes of a service item are represented as a set of attribute sets. An individual set of attributes may be represented as an instance of a class in the Java programming language, each attribute being a public field of that class. The class provides strong typing of both the set and the individual attributes. A service item can contain multiple instances of the same class with different attribute values as well as multiple instances of different classes. For example, an item may have multiple instances of a Name class, each giving the common name of the service in a different language, plus an instance of a Location class, an Owner class, and various service-specific classes.

Service Items are stored in the lookup service as instances of the ServiceItem class, as described below:

```
public class Service Item {
        public static final long ASSIGN_SERVICE_ID = 0;
        public long serviceID;
        public Object service;
        public Entry [] attributeSets;
}
```

The "serviceID" element is a numerical identifier of the service. The "service" element is an object representing the service or a stub facilitating access to the service, and the "attributeSets" element contains the array of attributes for the service.

Items in the lookup service are matched using instances of the ServiceTemplate class, and the ServiceItemFilter interface which are defined below:

```
public class Service Template {
        public static final long ANY_SERVICE_ID = 0;
        public long serviceID;
        public Class[] serviceTypes;
        public Entry[] attributeSetTemplates;
}
```

The "serviceTypes" element defines the types of the service. An item (item) matches a service template (tmpl) if item.serviceID equals tmpl.serviceID (or if tmpl.serviceID is zero) and

item.service is an instance of every type in tmpl.serviceTypes, and item.attributeSets contains at least one matching entry for each entry template in tmpl.attributeSetTemplates. Entry matching uses the following rule: an entry matches an entry template if the class of the entry is the same as, or a superclass of, the class of the template and every non-null field in the template equals the corresponding field of the entry. Every entry can be used to match more than one template. Both serviceTypes and attributeSetTemplates can be null in a service template.

The ServiceItemFilter interface defines the methods used by an object to apply additional matching criteria when searching for services on a lookup service. This filtering mechanism is particularly useful to entities that wish to extend the capabilities of the standard template matching schema previously discussed. For example, since template matching does not allow one to easily search for services based on a range of attribute values, this additional matching mechanism can be exploited by the entity to ask the managing object to find all registered printer services that have a resolution attribute between 300 and 1200 dpi. As seen below, the "check" method defines the implementation of the additional matching criteria to apply to a ServiceItem object found through standard template matching. This method takes the ServiceItem object as the sole argument, to test against the additional criteria. This method returns true if the input object satisfies the additional criteria, and false otherwise.

```
public interface ServiceItemFilter
{
        public boolean check(ServiceItem item);
}
```

The ServiceMatches class is used for the return value when looking up multiple items. The definition of this class follows:

```
public class ServiceMatches {
        public ServiceItem[] items;
        public int totalMatches;
}
```

The interface to the lookup service is defined by a ServiceRegistrar interface data structure. This interface is not a remote interface. Instead, each implementation of the lookup service exports proxy objects that implement the ServiceRegistrar interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. "Proxy objects" refer to objects that run in the client's address space and that facilitate access to the

lookup service. Methods are provided to register service items, find items that match a template, modify attributes of existing items, receive notifications when items are modified, and incrementally explore the collection of items along the three major axes: service ID, service type, and attributes. The definition of the interface follows:

```
public interface ServiceRegistrar {
        long REGISTRAR_SERVICE_ID = 1;
        ServiceLease register (Service Item item, long lease expiration) throws
RemoteException;
        Object lookup (ServiceTemplate tmpl)
                throws RemoteException
        ServiceMatches lookup(ServiceTemplate tmpl, int maxMatches)
                throws RemoteException;
        int addAttributes (ServiceTemplate tmpl, Entry [] attrSets)
                throws RemoteException;
        int modifyAttributes(ServiceTemplate tmpl, Entry [] attrSets)
                throws RemoteException;
        int TRANSITION_MATCH_NOMATCH = 1;
        int TRANSITION_NOMATCH_MATCH = 2;
        int TRANSITION_MATCH_MATCH = 3;
        EventRegID notify (Service Template tmpl,
                        int transition,
                        RemoteEventListener listener,
                        MarshalledObject handback,
                        long leaseExpiration)
                throws RemoteException;
        Class [] getEntryClasses (ServiceTemplate tmpl)
                throws RemoteException;
        Object [] getFieldValues (ServiceTemplate tmpl),
                        int setIndex,
                        String field)
                throws NoSuchFieldException, RemoteException;
        Class [] getServiceTypes (Service Template tmpl,
                                String packagePrefix
                throws RemoteException
```

This interface includes various methods, including the register method, the lookup method (single parameter form), the lookup method (two parameter form), the addAttributes method, the modifyAttributes method, the notify method, the getEntryClass method, the getFieldValues method, and the getServiceTypes method. The "register" method is used to register a new service and to re-register an existing service. This method is defined so that it can be used in an idempotent fashion. Specifically, if a call results in generation of an exception (in which case the

item might or might not have been registered), the caller can simply repeat the call with the same parameters.

To register a new service using the register method, item.ServiceID should be zero; if item.ServiceID does not equal any existing item's service object, then a new unique service id will be assigned and returned. The service id is unique over time with respect to this lookup service. If item.ServiceID does equal an existing item's service object, the existing item is deleted from the lookup service (even if it has different attributes), but that item's service id is reused for the newly registered item.

To re-register an existing service using the register method, item.ServiceID should be set to the same unique service id that was returned by the initial registration. If an item is already registered under the same service id, the existing item is deleted (even if it has different attributes or a different service instance). Note that service object equality is not checked in this case to allow for reasonable evolution of the service (e.g., the serialized form of the stub changes, or the service implements a new interface).

When an item is registered, duplicate attribute sets are eliminated in the stored representation of the item. Additionally, the registration is persistent across restarts of the lookup service.

The single-parameter form of the "lookup" method returns the service object (i.e., ServiceItem.service) from an item matching the template, or null if there is no match. If multiple items match the template, it is arbitrary as to which service object is returned. If the returned object cannot be deserialized, an exception is generated.

The two-parameter form of the "lookup" method returns at most maxMatches items matching the template, plus the total number of items that match the template. The return value is never null, and the returned items array is only null if maxMatches is zero. For each returned item, if the service object cannot be deserialized, the service field of the item is set to null and no exception is generated. Similarly, if an attribute set cannot be deserialized, that element of the attributeSets array is set to null and no exception is generated.

The "addAttributes" method adds the specified attribute sets (those that aren't duplicates of existing attribute sets) to all items matching the template. The number of items that were matched is returned. Note that this operation has no effect on existing attribute sets of the matching items. and the operation can be repeated in an idempotent manner.

The "modifyAttributes" method is used to modify existing attribute sets. The lengths of tmpl.attributeSetTemplates and attrSets must be equal or an exception is generated. For each item that matches the template, the item's attribute sets are modified as follows. For each array index *I*: if attrSets[*i*] is null, then every entry that matches tmpl.attributeSetTemplates[*i*] is deleted; otherwise, for every non-null field in attrSets [*I*], the value of that field is stored into the corresponding field of every entry that matches tmpl.attributeSettemplates[*i*]. The class of attrSets[*i*] must be the same as, or a superclass of, the class of tmpl.attributeSetTemplates [*I*], or an exception is generated. If the modification results in duplicate entries within an item, the duplicates are eliminated. The number of items that were matched is returned.

The "notify" method is used to register for event notification. The registration is leased, and the lease expiration request is exact. The concept of a lease is described in greater detail in U.S. Patent Application No. 09/044,923, entitled "Method and System for Leasing Storage," which has previously been incorporated by reference. The registration is persistent across restarts of the lookup service until the lease expires or is canceled. The event id in the returned EventRegId is unique at least with respect to all other active event registrations at this lookup service with different service templates of transitions.

While the event registration is in effect, a notification containing an indication of the event is sent to the specified listener whenever a register, lease cancellation or expiration, addAttributes, or modifyAttributes operation results in a service item changing state in a way that satisfied the template and transition combination. A list of the transitions follows, although other transitions may also be implemented within the scope of the present invention:

- TRANSITION_MATCH_NOMATCH: an event is sent when the changed item matches the template before the operation, but doesn't match the template after the operation (this includes deletion of the item).

- TRANSITION_NOMATCH_MATCH: an event is sent when the changed item doesn't match the template before the operation, (this includes not existing), but does match the template after the operation.

- TRANSITION_MATCH_MATCH: an event is sent when the changed item matches the template both before and after the operation.

The "getEntryClasses" method looks at all items that match the specified template, finds every entry among those items that either doesn't match any entry templates or is a subclass of at least one match entry template, and returns the set of the (most specific) classes of those entries. Duplicate classes are eliminated, and the order of classes within the returned array is arbitrary. Null (not a empty array) is returned if there are no such entries or no matching items. If a returned class cannot be deserialized, that element of the returned array is set to null and no exception is thrown.

The "getFieldValue" method identifies all items that match the specified template. This method returns the values of the items that match the specified template.

The "getServiceTypes" method looks at all items that match the specified template, and for every service object, this method finds every type (class or interface) of which the service object is an instance that is neither equal to, nor a superclass of, any of the service types in the template, and returns the set of all such types that start with the specified package prefix. Duplicate types are eliminated, and the order of types within the returned array is arbitrary. Null (not an empty array) is returned if there are no such types. If a returned type cannot be deserialized, that element of the returned array is set to null and no exception is thrown.

The Lookup Service Processing

Figures 3A and 3B depict a flowchart of the steps performed when a client, a program running on a particular device, makes use of the lookup service 212. Initially, the device on which the client runs is connected to the Jini distributed system (step 302). Next, the client sends a multi-cast packet containing code for communication with the client (step 304). In this step, the client is performing the discovery protocol as described in further detail in co-pending U.S. Patent Application No.09/044,945, entitled "Apparatus and Method for Providing Downloadable Code for Use in Communication With a Device in a Distributed System," which has previously been incorporated by reference.

After the client sends the multi-cast packet, the discovery server 214 receives the packet and uses the code contained in the packet to send a reference to the lookup service to the client (step 306). After the client receives the reference to the lookup service, the client is able to utilize the interface of the lookup service to either add a service, delete a service, access a service, or request notification when the lookup service is updated, as reflected by steps 308-326.

At some point during the processing of the client, it may decide to add a service to the lookup service (step 308). If it decides to add a service, the client adds a service to the lookup service by invoking the register method, which sends to the lookup service either an object representing the service or a stub containing code and data to facilitate access to the service (step 310). The addition of the stub to the lookup service is described in greater detail in co-pending U.S. Patent Application No. 09/044,826, entitled "Method and System for Facilitating Access to a Lookup Service," which has previously been incorporated by reference.

Next, the client may decide to delete one of its services from the lookup service (step 312). If a client decides to do so, the client deletes the service from the lookup service by canceling the service's lease with the lookup service (step 314). It should be noted that both the addition of a service and the deletion of a service are done dynamically and in a manner that does not prohibit other clients from using the lookup service while the update occurs.

At some point later in the processing of the client, the client may decide to access a service provided by the lookup service (step 316). If a client decides to access a service provided by the lookup service, the client accesses the service by invoking the lookup method, which retrieves from the lookup service either the object or the stub information for the service, and the client then either invokes methods on the object to use the service or uses the stub information to access the service (step 318). The step is described in greater detail in co-pending U.S. Patent Application No.09/044,826, entitled "Method and System for Facilitating Access to a Lookup Service, " which has previously been incorporated by reference.

The client may also request to be notified when an update occurs to the lookup service (step 320 in Fig. 3B). If a client wishes to be notified, the client invokes the notify method on the lookup service interface to register a callback routine with the lookup service (step 322). A "callback routine" is a function that is invoked when the lookup service is updated. Additionally, the notify method allows the client to register an object that will be passed back, via RMI, as a parameter to the callback function.

Next, if an event has occurred for which the client wants to be notified (step 324), the registered callback routine is invoked by the lookup service (step 326). In this step, the client is notified of the occurrence of the event and can take appropriate action. For example, if a service that the client was currently using has become unavailable, the client may store information so that it no longer uses it.

Figure 4 depicts a flow chart of the steps performed by the lookup service when performing event-related processing. Initially, the lookup service receives registrations from a number of clients interested in receiving notifications when particular events occur (step 402). In this step, the lookup service receives the registrations via invocation of the notification method on the lookup service interface and stores into a table, known as the event table, all of the associated information, such as an indication of the client to be notified, a reference to the callback routine, an object to be passed as a parameter to the callback routine, and an indication of the event in which the client is interested. It should be noted that a client may register to be notified upon the occurrence of an event, or the client may register for a third party to be notified. After receiving the registrations, the lookup service determines whether an event occurred such that at least one client has registered an interest in the event (step 404). The lookup service makes this determination by identifying when, for example, a new service has been added to the lookup service, an existing service has been deleted from the lookup service, or the attributes of a service have been modified. If such an event has not occurred, the event notification process of the lookup service remains in a wait state.

However, if an event has occurred, the lookup service determines all clients registered for notification for this event (step 406). The lookup service makes this determination by accessing the event table. Next, the lookup service invokes the callback routines registered for each client identified in step 406 (step 408). In this step, the event table contains a reference to the callback routine registered by each client, and the lookup service invokes each callback routine, passing the registered objects as parameters, to notify the clients of the occurrence of the event.

Alternative Embodiment

Since lookup services are generally located on a computer remote from the client, a query to a lookup service typically involves a remote call. Such remote calls are much more costly in terms of processing overhead and failure risk than are local procedure calls. This cost is magnified when a client must make frequent queries for multiple services. Such scenarios make it desirable for a client to internally store references to discovered lookup services and network services of interest. This approach permits the client to identify and access lookup services by simply searching a locally stored cache. It also provides the capability for a client to efficiently react to network failures by maintaining a redundant collection of network services. The

processing overhead required to perform these functions also makes it desirable to delegate the scheduling of them to a client lookup manager, and thereby free the client to perform other functions. FIG. 8 depicts a client computer in accordance with an alternate embodiment of the present invention. As shown in FIG. 8, computer 104, includes a memory 802, secondary storage device 804, CPU 806, input device 808, video display 810, and Java runtime system 316. Client computer 104 additionally includes a client 812, a client lookup manager 814 and a cache 815. The client lookup manager 814 is implemented by a client lookup manager class comprised of a helper utility class that any client 812 can use to create and populate a cache 815, and with which the client can register for notification of the availability of services of interest. A helper utility is a programming component that can be used during construction of Jini services and/or clients. Helper utilities are not remote. In other words, their methods typically do not execute on remote hosts. Consequently, they do not register with a lookup service and they are instantiated locally by devices wishing to employ them.

In accordance with the alternate embodiment, when the client lookup manager discovers a lookup service 212, the client lookup manager 814 evaluates each of the associated network services and stores references to network services of interest in the local cache 815. As shown in FIG. 9, when the client lookup manager receives a reference to a lookup service (step 900), it extracts the network service (step 910) and evaluates whether the service is of interest to the client (step 920). If the service is of interest, the client lookup manager adds the service to the cache in step 930 and process flows to step 940. If the service is not of interest, process immediately flows to step 940 where the client lookup manager determines whether there are more network services available in the lookup service. If there are more services available, program execution returns to step 910, where another network service is extracted. If there are not any more network services available, process flows to step 950 where the client lookup manager determines whether there are any remaining undiscovered lookup services. If there are undiscovered lookup services, program execution returns to step 900 and a new lookup service is discovered. In the event that all lookup services have been discovered, then the process of populating the cache is complete and program execution ends.

To perform the function of populating the cache as shown in step 920, the client lookup manager utilizes a complex filtering algorithm to extract any subset of network services desired by a client. As shown in FIG. 10, this process begins in step 1000 with a user inputting the

ServiceTemplate and ServiceItemFilter arguments described earlier. As shown in step 1010, a null reference input to the template argument is treated as the equivalent of inputting a ServiceTemplate constructed with all null arguments (all wildcards). That is, as shown in step 1030, the cache attempts to discover all services contained in a lookup service. Otherwise, the cache attempts to discover services contained in each lookup service that match the inputted criteria (step 1020). Next, the process performs the filtering specified by the ServiceItemFilter argument. In contrast to a null input to the ServiceTemplate as previously discussed, a null input to the filter argument (step 1040) has the opposite effect in that processing terminates and only template-matching will be employed to find the desired services. Otherwise, additional filtering in accordance with the inputted criteria will be employed (step 1050) and the cache will be populated accordingly.

Once the cache is populated, the client lookup manager handles all activities related to the cache including accessing, updating and deleting the cache. For example, as shown in FIG. 11, when the client desires access to a service reference, it transmits a request, via a local event to the client lookup manager (step 1100). In step 1110, the client lookup manager searches the cache to determine whether an instance of the requested service is stored in the cache. If the service is found (step 1120), the client lookup manager returns an instance of the requested service in step 1130 and the process ends. If, the service is not found, the client lookup manager returns a null reference in step 1140, and the process terminates. Depending on the specific implementation of the client lookup manager, the process can either re-institute a new search, wait for a new query request from the client, or re-query the associated lookup services.

As shown in FIG. 12, if a requested service reference is not found in the cache or if an insufficient number of references is retrieved (in the case of a client request for multiple references), the client lookup manager can re-query the associated lookup services. As shown in step 1200, this form of lookup takes as input an integer argument (MaxMatches) that represents the maximum number of matches that should be returned. In other words, the object returned by this method will contain no more than that number of service references, although it may contain less. This method also takes an integer argument (WAIT) that indicates the maximum amount of time the process is to wait before returning the identified service references. This argument prevents the client lookup manager from attempting to discover a requested number of MaxMatches for an infinite period of time. This feature is particularly useful to a user who is

interested in choosing a service from a list of possible candidates instead of receiving a single service.

Once the ServiceTemplate, ServiceItemFilter, MaxMatches, and WAIT arguments are input, the process proceeds to step 1210 where the method accesses a lookup service. In step 1220, the method retrieves a service reference from the lookup service and in step 1230, determines whether the retrieved service reference is a qualifying service. If a qualifying service is found, the process proceeds to step 1240 and an array of ServiceItem objects is recorded. Otherwise, the process proceeds to step1250. After an array of ServiceItem objects is recorded in step 1240, the process then compares the number of identified matches with the MaxMatches value previously input (step 1260). If the number of identified matches equals the MaxMatches value, the process returns the identified array of ServiceItem objects in step 1270 and the process terminates. Otherwise, the process proceeds to step1250. As indicated earlier, this process will wait a predetermined period of time for the client lookup manager to identify the requested number of qualifying services. But, it returns immediately upon discovering the requested number of service references. In step 1250, the process determines whether any more unexamined services exist inside the lookup service. If there are more, process flows to step 1220 and another service reference is retrieved. If there are no more services in the lookup service, process flows to step 1280 to determine whether any more unexamined lookup services exist. If there are more lookup services, the process continues back to step 1210 to access another lookup service. If there are no more unexamined lookup services in step 1280, the process proceeds to step 1290 where the method determines whether the WAIT argument has been invoked (WAIT>=0) and if so, whether the duration has been exceeded.

The "wait" feature is quite useful to entities that cannot proceed until such a service of interest is found. As shown in FIG. 12, entities wishing to employ this feature must input a positive value to the WAIT argument which represents the number of milliseconds to wait for the desired service or services to be discovered. If a non-positive value is input to this argument, then this method will not wait. It will simply query all available lookup services once, and return the array of identified ServiceItem objects. If WAIT has been invoked and the elapsed time exceeds the WAIT duration, the process continues to step 1270, returns the array of ServiceItem objects found thus far and then the process terminates. On the other hand, if WAIT has been invoked and the duration has not been exceeded, the process continues to step 1291 and the client lookup manager registers with the event mechanism of the discovered lookup services. The event mechanism requests the lookup services notify the client lookup manager when one of the lookup service's associated networks services changes. When a service changes, the client lookup

manager determines whether the new service is a service of interest in step 1292. If the service is of interest, it is recorded in the ServiceItem array (step 1293). Otherwise, execution flows to step 1296 where the process determines whether the elapsed time is less than or equal to the WAIT duration. If it is, the client lookup manager waits to be notified of a new service. If the time has expired, the client lookup manager returns the array of ServiceItem objects found thus far in step 1295. Once a service is recorded in the ServiceItem array (step 1293) the client lookup manager determines whether the number of ServiceMatches is less than MaxMatches. If it is, execution flows to step 1296 where the process determines whether the elapsed time is less than or equal to the WAIT duration. If the number of ServiceMatches is equal to MaxMatches, then the desired number of services has been found and process flows to step 1295 where the identified array is returned. It should be obvious to those skilled in the art that instead of an array of objects, this process could return an instance of an object without departing from the scope of this process. It should also be obvious to those skilled in the art that instead of returning when the total number of references equals the maxMatches object, this method could instead return once a minimum number of references (minMatches) have been found.

Just as the client may request that it be notified of state changes in network services occurring within each lookup service, as previously described in FIG. 4, so too can the cache request that it be notified. In other words, the client and the cache can separately be notified of the same or different events after receiving notification from a lookup service. FIG. 13 shows the steps performed by the client lookup manager when either the cache or the client has requested event notification. First, as shown in step 1310, the lookup service observes a state change in a registered service. Next, the lookup service determines, in step 1320, whether the event satisfies matching criteria specified by one of its associated clients. If it does, the lookup service transmits the event notification to the requesting client (step 1330) via a remote event. If the event does not satisfy matching criteria, the process ends. Next, the cache (step 1350) determines whether the reported event necessitates a cache update. More specifically, since the cache may receive multiple events corresponding to the same service, it must first determine whether it has previously been notified of this particular event. For example, a particular service may be registered with more than one lookup service or multiple configurations of the same service can be registered with a lookup service. When a lookup service reports an event to a client, the lookup service has no way of knowing whether the event has previously been reported to the client via another lookup service. If the cache has not previously been notified of this event, the cache is updated in step 1360. Otherwise, the process ends (note: if the cache has been previously notified, then the client has also previously been notified). Once the cache has been updated, it

next determines whether the client is required to be notified as well (step 1370). If the client requested notification, a notification message is sent via local event to the client in step 1380 and the process terminates. This "many-to-one" relationship between the events received by the cache 815 and the events sent by the cache hides from the client 812, the lookup services with which the cache 815 interacts. For many entities that use the cache's event mechanism to interact with the cache's discovered services, knowledge of the number of distinct service references, as well as identification of the lookup services with which those references are registered, is of no interest. Such entities typically are interested only in acquiring a reference -- not all references to the services it needs to do its job. For entities which are interested in this additional information, the cache 815 provides methods separate from the event mechanism for obtaining that information.

The Conference Room Example

Described below is an example use of the lookup service 212 with respect to a conference room 500 as depicted in Fig. 5. The conference room 500 contains a printer 502, a fax machine 504, a computer 506, a projection screen 508, and a storage device 510, all interconnected by a network 512. The computer 506 contains a lookup service 518 that defines all services made available by the conference room Djinn. In this example, the user enters the conference room 500 with laptop 514. Such a situation occurs when a user with a laptop enters the conference room 500 for a meeting. If the user wishes to make use of the services provided by various devices in the conference room 500, the user must join the Djinn of the conference room. In order to do so, the user connects the laptop to the network 512 and runs a program, client 516, to perform the boot and join (or discovery) process, as described above. After the completion of the boot and join process, the client 516 (e.g., a browser) receives a reference to the lookup service 518 defining the services provided by the conference room Djinn. Using this reference, the client 516 downloads an applet from the lookup service 518 that displays a screen 600, as shown in Fig. 6.

Fig. 6 depicts screen 600 displaying the available services of the lookup service 518 represented as various icons, including a printer 602, a fax 604, a file system 606, and a projection screen 608. The screen 600 also displays an add service button 610. When the printer icon 602 is selected, the stub code (i.e., service item) for the printer is downloaded to the client so that the client may make use of the printer. Similarly, selection of the fax icon 604, the file system icon 606, or the projection screen icon 608 downloads the appropriate stub information so that the client can use those services. The file system icon 606 represents the file system of the computer 506.

The user, however, may want to add a service and thus selects the add service button 610. Upon selection of the add service button 610, a screen 700 is presented to the user as shown in Fig. 7. The screen 700 depicts the available services of the laptop 514 which the user may add to the Djinn. For example, the available services on the laptop 514 may include a Java space 702, the laptop file system 704, a database 706, and a dictionary 708. The user may select any of these services, which are then automatically added to the lookup service and made available to other users of the Djinn.

Although methods and systems consistent with the present invention have been described with reference to a preferred embodiment thereof, those skilled in the art will know of various changes in form and detail which may be made without departing from the spirit and scope of the present invention as defined in the appended claims.